Performance Portability of Multi-Material Kernels

István Z. Reguly
Faculty of Information Technology and Bionics
Pázmány Péter Catholic University
Budapest, Hungary
reguly.istvan@itk.ppke.hu

***Abstract*—Trying to improve performance, portability, and productivity of an application presents non-trivial trade-offs, which are often difficult to quantify. Recent work has developed metrics for performance portability, as well some aspects of productivity - in this case study, we present a set of challenging computational kernels and their implementations from the domain of multi-material simulations, and evaluate them using these metrics. Three key kernels are implemented using OpenMP, OpenMP offload, OpenACC, CUDA, SYCL, and KOKKOS, and tested on ARM ThunderX2, IBM Power 9, Intel KNL, Broadwell, and Skylake CPUs, as well as NVIDIA P100 and V100 GPUs. We also consider the choice of compilers, evaluating LLVM/Clang, GCC, PGI, Intel, IBM XL, and Cray compilers, where available. We present a detailed performance analysis, calculate performance portability and code divergence metrics, contrasting performance, portability, and productivity.**

***Index Terms*—OpenMP, OpenACC, CUDA, SYCL, KOKKOS, benchmarking, performance**

I. Introduction

Performance portability is now perhaps more of an issue than ever. With Moore's law slowing down [1], and Artificial Intelligence becoming one of the most important drivers of high performance computing (HPC) architectures, we are facing a staggering diversity in HPC hardware. With no clear successor, CMOS technology is here to stay for at least a decade, and with transistor sizes no longer becoming smaller at the rates they used to, we expect further specialisation of architectures, targeting various narrow application domains - such as the instructions and circuits already present in mainstream CPUs (Intel's VNNI [2]) and GPUs (NVIDIA's Tensor Cores [3]).

The US DoE is further accelerating this diversification - with two pre-exascale machines (Summit and Sierra) already in place that use IBM CPUs, and NVIDIA GPUs, a number of large Intel CPU systems, and the planned exascale systems, one from Intel utilising their as of yet unseen Xe GPUs, and another from AMD, also utilising CPUs and GPUs. Almost all of these systems use CPU+accelerator architectures, which in some sense are similar, but in terms of programmability, they are quite different. At the same time, CPU-only systems are not going away either, given the vast amounts of legacy code - particularly at institutions looking after nuclear stockpiles, who are some of the main users of multi-material simulations. Governments and institutions in Europe are also more conservative in their choice of HPC platforms - there are many more large CPU-only systems there.

This increasing diversity further exacerbates the performance, portability, and productivity challenge. While there are vast performance gains to be had by fine-tuning algorithms for a particular architecture, there are too many options, and the lifetime of these codes tend to be far longer than the hardware itself. Maintaining and updating different specialised versions of code for the various architectures as they come and go is simply untenable above a certain size.

There are a staggering number of approaches trying to address this issue - in the general case there is always a trade-off between performance, portability and productivity. Parallel programming approaches are being extended - for example OpenMP offload support [4], [5], which even though supports CPUs and GPUs in the same framework, still has separate pragmas for the two. OpenACC [6] was the first widely adopted pragma-based approach to program GPUs, but to this day its main focus is NVIDIA GPUs. The NVIDIA-only CUDA [7] programming extension of C/C++ has developed a large software ecosystem, and is well supported, but obviously not portable. Efforts for a lower-level, but more portable compute abstraction include OpenCL [8], which has failed to gain traction in the high performance computing market. OpenCL also did not really deliver performance portability - particularly the low-level optimisations have to be specialised or re-implemented for different targets. SYCL [9] is a (re-)incarnation of OpenCL with a modern C++ API, which promises much better programmability, and Intel is adopting it as OneAPI; a single programming approach for all their devices. How well it is received and adopted remains to be seen. KOKKOS [10] and RAJA [11] are C++ portability layers designed to map to OpenMP and CUDA (allowing for further lower-level models as well), which do narrow the supported set of algorithms, but provide reasonable performance portability. Going even further, Domain Specific Languages (DSLs) focus on a particular set of algorithms occurring in a given problem domain, but are able deliver true performance portability.

Performance portability is therefore still a big, unsolved problem - even its quantification. Recent work by Pennycook et al. [12] has developed a measure for this, that is currently the best we have - it gives a single number for how well a given code runs on different platforms. Another big challenge is quantifying productivity; how much effort goes into developing and maintaining a code base. Harrell et al. [13] published a study on this, trying to identify different factors involved in the process. A recently released tool, the Code

Base Investigator [14] provides a script to help quantify how much specialised code exists in a codebase to support different platforms and optimisations.

Multi-material simulations are an important class of applications where multiple different materials mix in the same simulation domain. In its structure, it is a sparse problem, meaning that in any given small volume of space (e.g. a given discretisation's cell) there may be one, or a few materials present, out of a large number of cells and possible materials. The implication is that storing state variables for all possible materials in all cells is extremely wasteful, and usually requires prohibitively large amounts of memory; therefore in most cases an irregular "compact" data structure is used, which only stores non-zero values. Prior work has studied these data structures, and their performance [15], [16], and our own work is based on these - we study three typical algorithmic patterns, their optimisations, and their performance with various parallelisations, compilers, and hardware.

The contributions of this paper are as follows:

1) We develop OpenMP, OpenMP offload, OpenACC, CUDA, SYCL, and KOKKOS implementations for three multi-material kernels.
2) We evaluate performance on Intel, IBM, and ARM CPUs, as well as NVIDIA GPUs, with a number of different compilers, including GCC, LLVM/Clang, Intel, Cray, XL, and PGI.
3) We discuss the performance, portability, and productivity implications and measures following [12], [14].

All source codes used in this paper and the performance data are available at [17].

The rest of the paper is organised as follows: Section II presents the studied algorithms and their parallelisations, Section III briefly summarises prior work on the metrics for performance, portability and productivity, based on [13], [14]. Section IV presents the performance results, and Section V presents the discussion. Finally, Section VI draws conclusions.

II. Multi-Material Algorithms

Multi-material algorithms are mainly used in multiphysics applications, where state variables are associated with each material in each cell. Here, we use density $\rho_{C,m}$, temperature $t_{C,m}$, pressure $p_{C,m}$, and volume $V_{C,m}$, which are defined for all cells C and materials m. Volume is generally stored as the volume of the cell V_C and the fractional volume of constituent materials $V_{fC,m}$ (with their sum in any given cell being 1). Here, we consider three key algorithms, as described in [15]:

1) Algorithm 1: Compute the weighted average density of materials in each cell.
2) Algorithm 2: Compute the pressure in each material contained in each cell using the ideal gas law.
3) Algorithm 3: Compute the weighted average density of each material over neighbouring cells.

The simplest data storage scheme, commonly referred to as "full matrix", has entries for all possible material-cell combinations, and is commonly represented as a matrix of size $N_C \times N_m$. When there are a large number of materials, but each cell contains only a small number of different materials, this storage is prohibitively wasteful, necessitating "compact" storage formats. Compact storage maintains arrays of size C for the state variables, as well as for $materials$ - an index in each cell for the material contained, if there is only one, and an index into a separate $frac$ data structure if there are multiple. The $materials$ array is commonly set up to use positive indexes for material IDs for pure cells, and negative indexes, which when multiplied by -1, give the position within the $frac$ data structure for the first material contained within the cell. For details, see [15].

There are two common ways to store data for mixed cells in the $frac$ data structure: linked lists, and compressed sparse row (CSR). In both cases, an entry will contain data about the material index, the original cell index, the state variables, and the fractional volume. With linked lists, each fraction contains a pointer to the next fraction, and with CSR fractions are grouped and stored contiguously, therefore index ranges are provided. The linked list version is easier to insert into or remove from, whereas the CSR version uses slightly less memory, and allows for easy fission of some loops into pure and mixed parts (discussed below).

We present pseudocode for the three algorithms here, but for brevity we only show their "full matrix" formulation - the ones using compact storage have extra logic for handling the iteration over materials in a mixed cell, making the code and the algorithm much more complicated, especially for Algorithm 3. For full details, please refer to [15], or the actual implementations available at [17].

Algorithm 1 Algorithm 1: weighted average density of materials in each cell [15].

```
for all cells, $C$, in the mesh do
    $ave = 0$
    for all material IDs, m, in the problem do
        if $V_{fC,m} > 0.0$ then
            $ave += \rho_{C,m} * V_{fC,m}$
        end if
    end for
    $\rho_{aveC} = ave/V_C$
end for
```

Algorithm 2 Algorithm 2: pressure of each material in each cell [15].

```
for all cells, $C$, in the mesh do
    for all material IDs, m, in the problem do
        if $V_{fC,m} > 0.0$ then
            $p_{C,m} = (n_m * \rho_{C,m} * t_{C,m})/V_{fC,m}$
        end if
    end for
end for
```

When values are stored in a compact data structure, there is further branching in Algorithms 1-3, for pure cells, where

Algorithm 3 Algorithm 3: weighted average density of each material over neighbouring cells [15].

```
for all cells, $C$, in the mesh do
    for all neighbours i do
        $sqrdist_i = (x_C - x_i)^2$
    end for
    for all material IDs, m, in the problem do
        if $V_{fC,m} > 0.0$ then
            $\rho_{sum} = 0$
            $N_n = 0$
            for all neighbours i do
                if $V_{fi,m} > 0.0$ then
                    $\rho_{sum} += \rho_{i,m}/sqrdist_i$
                    $N_n += 1$
                end if
            end for
            $\rho_{aveC,m} = \rho_{sum}/N_n$
        end if
    end for
end for
```

the state variables are simply indexed with the cell index, and for mixed cells, where the $frac$ structure has to be traversed. This introduces further complexity into the code and due to the divergence, it also maps poorly to vector architectures. Algorithms 1 and 2 can be formulated in two different ways when using the compact data structure: (1) a branch in the loop body, accessing values differently for pure and mixed cells (referred to as "fusion version"), or (2) one flat loop for pure cells (potentially performing useless computations on mixed cells to avoid branching), and one flat loop across mixed cells (referred to as "fission version"). For Algorithm 2, the second formulation is particularly efficient, as there are no indirections or branching needed, making it a largely bandwidth-bound kernel, at the cost of some extra computations, proportional to the number of mixed cells.

In summary, we have three algorithms, two variants for the $frac$ data storage; linked lists and CSR, and fusion/fission variants of Algorithms 1 and 2.

We evaluate two test problems, both of size 3000^2 on a Cartesian grid, with the first (Problem 1) following the structured material layout in concentric squares as discussed in [15] (their example is 1000^2 - we simply replicate it 3 times in each direction), which has 95% pure cells, 4.9% 2-material cells, 0.06% 3-material, and 0.04% 4-material cells. The second test case (Problem 2) is randomly generated (with the same seed for different runs), and reflects a more complex scenario, with 60% pure cells, 30% 2-material, 5% 3-material, and 5% 4-material cells.

A. Implementation and Test Problems

The parallelisation of these algorithms across different cells is straightforward, as there are no dependencies between them. While it is also possible to parallelise across different materials within any given mixed cell (with an extra reduction in Algorithm 1), it is inefficient to do so, as mixed cells only contain up to 4 materials in our setup. The only exception is Algorithm 2, where the fission + CSR variant makes it trivial to parallelise across all cell-material combinations. The developed OpenMP, OpenMP offload, OpenACC, CUDA, SYCL, and KOKKOS implementations therefore explicitly parallelise across cells, and iterations across materials in mixed cells is done sequentially (except for the aforementioned variant of Algorithm 3).

We intentionally developed the different variants with as much code reuse as possible. The code paths for the various parallelisations are enabled using preprocessor macros, and are in a single `.cpp` file for OpenMP, OpenMP offload, OpenACC, and SYCL, changing only the surrounding loop structures, not the loop body, and a separate `.cu` file for CUDA, as well as a separate file for KOKKOS, due to the need for using the parentheses operator for accessing data. This arguably makes some of the code quite difficult to read. For details, see `compact.cpp` in [17].

III. Performance, Portability, Productivity Metrics

In this section, we summarize the key points and the metrics presented in [12] and [13].

The definition of Performance Portability, as defined in Pennycook et al. [12]: "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set.". The metric described in the paper gives a single value, $P(a, p, H)$, as a function of a given application a, running a given problem p, on a set of hardware/software platforms of interest H (where $|H|$ is the number of platforms).

$$P(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

which is the harmonic mean of performance efficiencies $e_i(a,p)$ on each platform. There are two common metrics for performance efficiency on a given hardware (e_i): as a fraction of some peak theoretical performance (e.g. bandwidth of computational throughput), or as a fraction of "best known performance" on the given platform. In our work, we use the first version, and define efficiency on a platform as a fraction of theoretical peak bandwidth achieved, as our multi-material kernels have a very low computational intensity, and are therefore bound by bandwidth and latency - unfortunately latency is rather difficult to measure quantitatively.

This performance portability metric has the property that it is zero if the application does not run on any given platform of interest, and it increases with any performance increases on any of the platforms. However, it does not consider productivity - in the extreme case, it still considers completely separate implementations and optimisations of the same applications as one. For obvious reasons, working with such a codebase is very unproductive. A “code divergence” metric was proposed by Harrell et al. [13], which quantifies the difference, relying on the number of different lines of code, between variants targeting different platforms. Code divergence D on a set of code variants is defined as follows:

$$D(A) = \binom{|A|}{2}^{-1} \sum_{\{a_i,a_j\}\subset A} d(a_i, a_j), \qquad (2)$$

giving the average pairwise distances between all the variants in A (where $|A|$ is the number of variants). d is defined as the change in the number of source lines of code (SLOC):

$$d(a, b) = \frac{|SLOC(a) - SLOC(b)|}{min(SLOC(a), SLOC(b))}. \qquad (3)$$

We use the Code Base Investigator tool [14] to calculate this metric for our applications.

IV. Performance Results

We evaluate performance on a number of different platforms, parallel programming approaches, and compilers;

1) ARM: using a single CPU socket in a node of Isambard, a Cray XC50 system, with 32-core Cavium ThunderX2 processors running at 2.1 GHz, and 128 GB of DDR4-2666. We evaluate stand-alone OpenMP, and KOKKOS with OpenMP, with 1,2,4,8 threads per core, using the Cray compilers (8.7.9), GCC (8.2.0), and LLVM/Clang (9.0), each with its own implementation of OpenMP.
2) Power 9: a single socket IBM Power 9 CPU with 10 cores, running at 3.8 GHz, with 128 GB of DDR4-2666, running RHEL 7.5. We evaluate stand-alone OpenMP, and KOKKOS with OpenMP, with 1,2,4,8 threads per core, using the GCC (7.3.1) and LLVM/Clang (9.0) compilers, each with its own implementation of OpenMP.
3) Intel Broadwell (BDW): a single socket of Intel Xeon E5-2660 v4 CPU with 14 cores per socket, running at 2.0GHz, and 64 GB of DDR4-2666, running Debian 9. We use 2 threads per core with stand-alone OpenMP, and KOKKOS with OpenMP, and compile with Intel 2018.2, GCC (8.1.0), and LLVM/Clang (9.0), each with its own implementation of OpenMP. We also evaluate performance with SYCL, with Intel’s LLVM implementation and ComputeCpp.
4) Intel Skylake (SL): a single socket of Intel Xeon Silver 4116 CPU with 12 cores per socket, running at 2.10GHz, and 96 GB of DDR4-2666, running Debian 9. We use 2 threads per core with stand-alone OpenMP, and KOKKOS with OpenMP, and compile with Intel 2018.2, GCC (8.1.0), and LLVM/Clang (9.0), each with its own implementation of OpenMP. We also evaluate performance with SYCL, with Intel’s LLVM implementation and ComputeCpp.
5) Intel KNL: an Intel Xeon Phi x7210 running at 1.3 GHz, and allocating memory in the 16 GB MCDRAM, running Debian 9. We evaluate 1,2,4 threads per core with stand-alone OpenMP, and KOKKOS with OpenMP, and compile with Intel 2018.2, GCC (8.1.0), and LLVM/Clang (9.0), each with its own implementation of OpenMP.
6) NVIDIA P100: a PCI-e card with 16 GB memory, running at 1.328 GHz, with CUDA 9.2. We evaluate OpenACC with PGI compilers 18.10, OpenMP offload in LLVM/Clang (9.0), CUDA compiled with nvcc (9.2) or LLVM/Clang (9.0), and KOKKOS compiled with nvcc. We also evaluate performance with SYCL, with hipSYCL [18] and ComputeCpp [19].
7) NVIDIA V100: a PCI-e card with 16 GB memory, running at 1.245 GHz, with CUDA 9.2. We evaluate OpenACC with PGI compilers 18.10, OpenMP offload in LLVM/Clang (9.0), and CUDA compiled with nvcc (9.2) or LLVM/Clang (9.0), and KOKKOS compiled with nvcc. We also evaluate performance with SYCL, with hipSYCL and ComputeCpp.

The maximum theoretical bandwidth for each platform is given in Table I. First, we present results from the best algorithmic variant with each platform+compiler combination in Figures 1 and 2, running Problems 1 and 2. Results with KOKKOS are shown separately in Figure 3, and discussed in Section IV-A. Performance is shown as the fraction of peak bandwidth achieved.

Algorithms 1 and 2 are clearly bandwidth-bound, and they achieve a large fraction of peak with most platforms and compilers. The best fraction of peak is consistently achieved on the ARM system, and the second-best on Power. On Intel systems, and the KNL in particular, the Intel compilers outperform others, but overall utilisation is lower compared to other hardware - however, while on ARM and Power, efficiency goes down slightly when moving from Problem 1 to 2, efficiency on Intel goes up (KNL+Intel especially).

Algorithm 3 has very low utilisation, due to its irregularity there are many branch mispredictions, and it is quite unfriendly to vectorisation. The Intel compiler is capable of vectorising computations across adjacent cells, while the others were not - vectorising across different materials in the same cells is inefficient, particularly with longer vectors, due to the low trip count ($<= 4$). Problem 1 has a slightly structured layout of materials, and few mixed cells (5%), whereas Problem 2 is fully random and has 40% mixed cells. The difference shows in performance particularly on Intel hardware; on the Broadwell platform, vectorisation (256 bit AVX2) does improve performance and on Problem 1, it achieves a reasonable fraction of peak (27%). On Skylake, vectorisation degrades performance, and the low utilisation of the double width vectors (512 bit AVX512) significantly affects overall efficiency as well. On the KNL (which also has 512 bit vectors) vectorisation is needed, but even so, efficiency is low. The Intel compilers’ advantage on these platforms is clearly in its ability to vectorise - without vectorisation (on Skylake), or when vectorisation efficiency is lost due to irregularity, performance with other compilers and parallelisations is quite close. ARM and Power9 both have 128 bit vector units, and therefore are much less affected by a lack of vectorisation,

TABLE I
THEORETICAL BANDWIDTH NUMBERS FOR THE TESTED PLATFORMS (SINGLE SOCKET) IN GB/S

ARM	Power 9	Broadwell	Skylake	KNL	P100	V100	R VII
144	170	71	107	490	732	900	1024

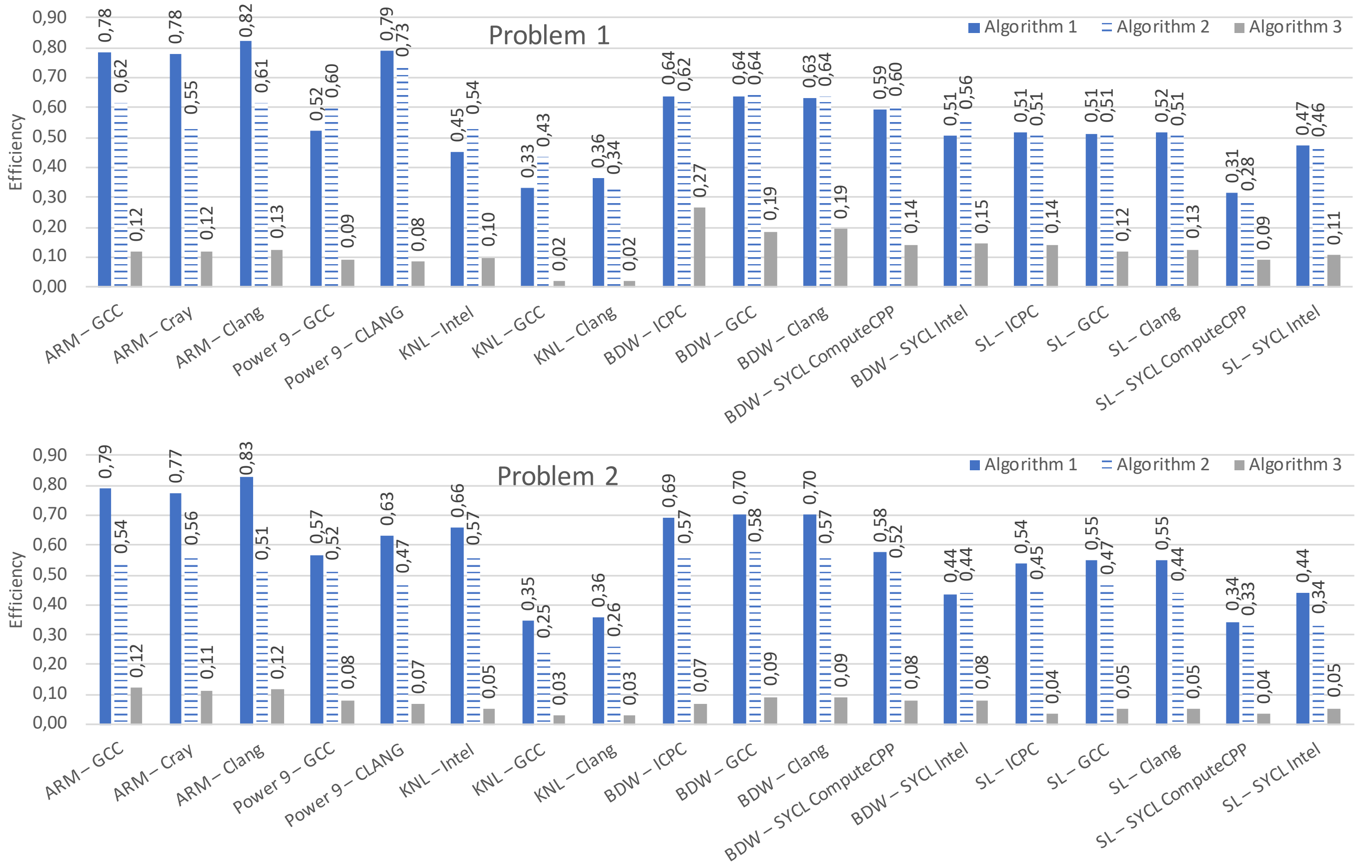

Fig. 1. Performance efficiency (fraction of peak performance) of the 3 multi-material algorithms on different CPU architectures, parallelisations and compilers, on Problem 1 and 2.

and they have less aggressive branch prediction as well, which means that while on Problem 1 they achieve lower efficiency than Intel hardware, there is very little loss (5-15%) when moving to Problem 2 compared to Intel's (40-75%), where they achieve higher efficiency in comparison.

The best performing algorithmic variant varies widely with architectures and compilers.

On ARM, Algorithm 1 and Problem 1 perform best with a CSR+fission variant by a wide margin (15%) with GCC and Clang, whereas with Cray it is 17% slower than the other variants - but overall the best performing ones are within 6% of each other. Running Problem 2, all three compiler perform best with linked lists+fusion (again within 6% of each other). On Algorithm 2 and Problem 1, there is very little difference, but on Problem 2, the CSR+fission variant outperforms others by 20-35% (with up to 10% difference between compilers). With Algorithm 3 and Problem 1, there is again very little difference, but moving to Problem 2, linked lists+fusion outperforms others 20-30% (compilers are within 6% of each other). Overall, there is no clearly better compiler, each is best on a different algorithm.

On Power, the variations are significantly lower than on ARM, but we see the same data structures performing best on the same algorithms and problems. GCC outperforms Clang by 6% overall. Based on prior experience [20] the IBM XL compilers generate more efficient code, however, we have been unable to configure the OpenMP environment (for thread binding) on the test machine to achieve reasonable performance, therefore we omit those results.

On Intel platforms, the Intel compilers far outperform all other compilers - on the KNL especially. We again see different problems performing best with different algorithmic variants. On the KNL, Clang and GCC perform within 5%, but are slower than Intel by 51% on average. On Broadwell the gap narrows significantly, and they perform within 7% of each other for Algorithms 1 and 2, but for Algorithm 3, Intel is 38% faster on Problem 1, but 25% slower on Problem 2 when vectorisation is enabled. Currently, SYCL implementation s are 5-15% slower than OpenMP ones, but they exhibit the same algorithmic variant preferences as seen

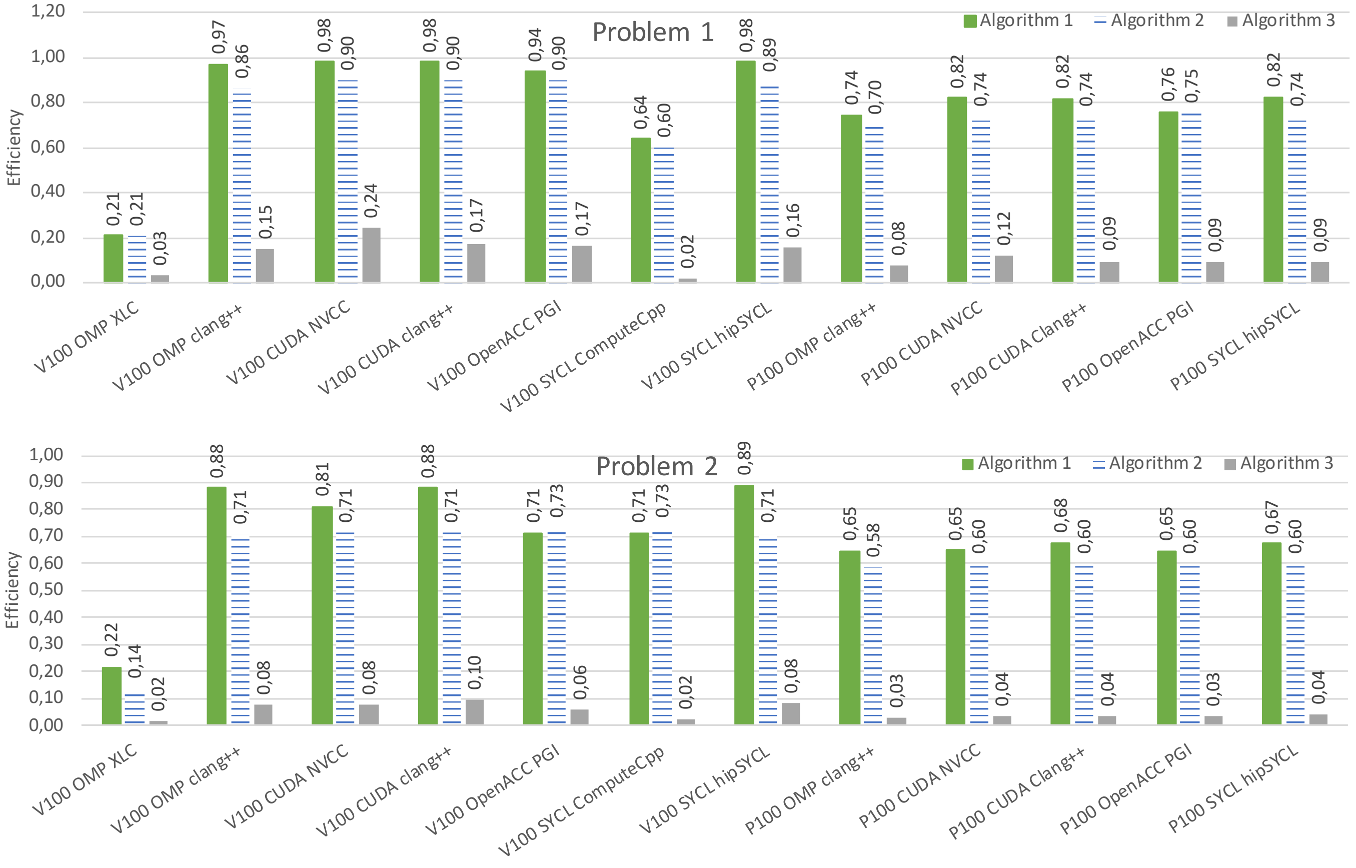

Fig. 2. Performance efficiency (fraction of peak performance) of the 3 multi-material algorithms on different GPU architectures, parallelisations and compilers, on Problem 1 and 2.

with OpenMP. Skylake behaves similarly to Broadwell, except even on Problem 1, Algorithm 3 performs worse with SIMD.

Moving to GPUs, we see very good bandwidth utilisation for Algorithms 1 and 2 for most compilers, except XL+OpenMP offload, and the ComputeCpp SYCL implementation (which at this stage is highly experimental). While on CPUs, efficiency of Algorithms 1 and 2 largely improved when moving from Problem 1 to 2, there is a slight reduction observed here (around 10%). The V100 overall achieves a higher efficiency, on average by 15%. Algorithm 3 once again achieves a small fraction of peak performance, with the best being CUDA code compiled with NVCC - 25% on the V100 and 12% on the P100. Most other compilers perform within 10% of each other. Moving to Problem 2 again significantly reduces efficiency due to the additional divergence - efficiency is between 6-8% on the V100 and 3-4% on the P100 - with the notable exception of CUDA compiled with Clang outperforming others by 20% (likely due to better ILP).

The performance of different data structures and algorithmic variants once again varies significantly - for Algorithm 1 and Problem 1, CSR+fission performs best (7-20%), but with Problem 2, it's linked lists+fusion (5-15%). For Algorithm 2, CSR+fission performs best in both problems by a margin of 5-30%, and with Algorithm 3, linked lists+fusion performs better on both problems (15-20%).

A. *KOKKOS results*

We test the OpenMP and the CUDA capabilities in KOKKOS to run the same source code on the various platforms - as previously, we compile with GCC and Clang, and Cray/Intel where available. Overall, the behaviour of the code follows the hand-written OpenMP and CUDA implementations, with only a slight degradation in performance, as reported in Figure 3. As there is slightly less control over vectorisation when using KOKKOS, there is very little difference between compilers - especially with Intel, which does outperform others in the hand-written OpenMP implementations. A notable difference is on the Intel KNL platform, where despite enabling the experimental HBW (high-bandwidth memory) support, we still see low efficiency with Intel and GCC compilers - with Clang however, performance exceeds that of the hand-written OpenMP implementation.

V. Discussion

The results reported show that performance depends on a large number of factors, that are often at odds with each other. Diverse algorithms make up an application, and each of these algorithms perform differently given different data structures, optimisations, parallelisations, and compilers - all of which may also depend on the input problem.

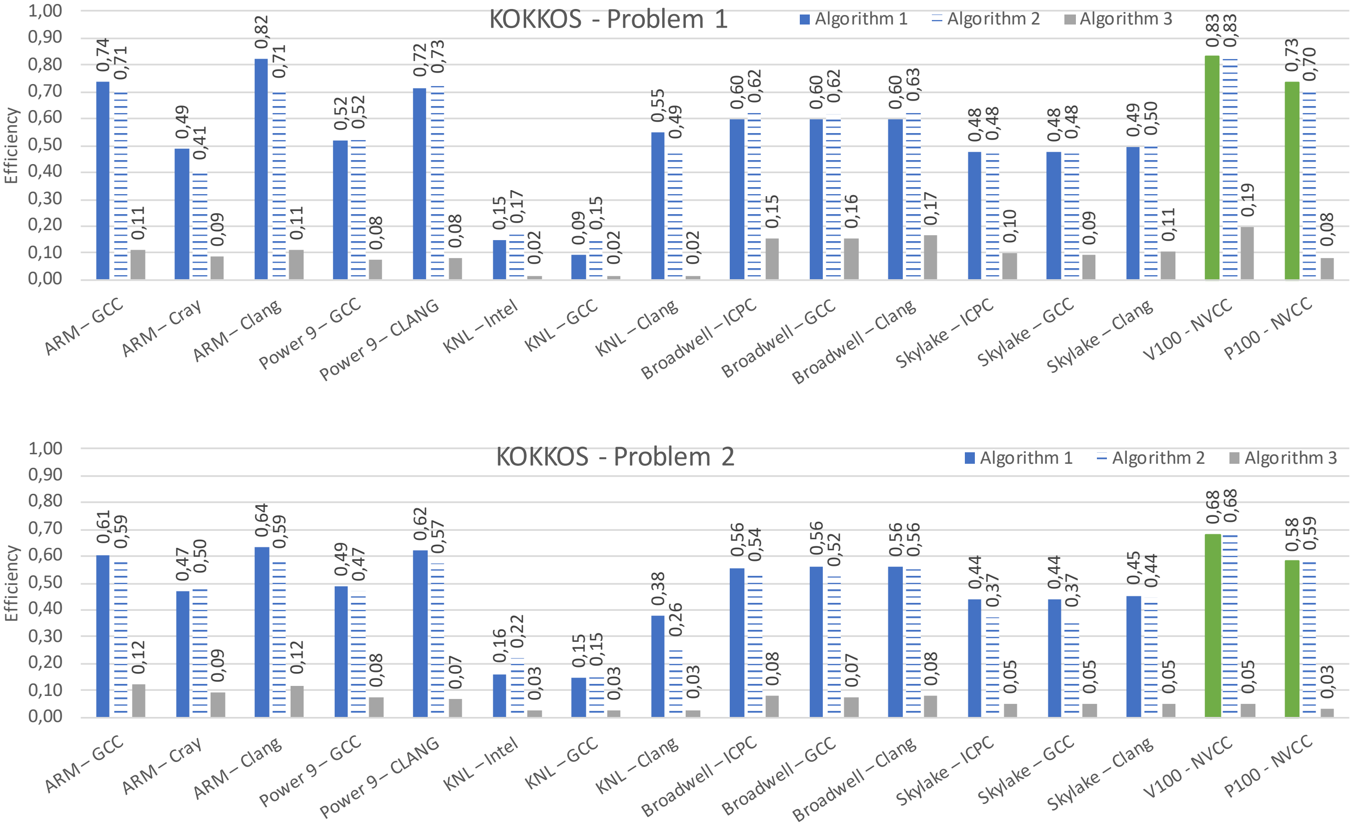

Fig. 3. Performance efficiency (fraction of peak performance) of the 3 multi-material algorithms on different architecture running KOKKOS, on Problem 1 and 2.

TABLE II
BEST EFFICIENCY (FRACTION OF PEAK PERFORMANCE) ON EACH PLATFORM FOR THE TWO PROBLEMS, AND THE PERFORMANCE PORTABILITY METRIC

	ARM	Power 9	Broadwell	Skylake	KNL	V100	P100	R VII	P
Method+Compiler	OMP+Clang	OMP+Clang	OMP+Intel	OMP+Intel	OMP+Intel	CUDA+NVCC	CUDA+Clang	SYCL+hipSYCL	
Problem 1	0.52	0.54	0.36	0.51	0.39	0.71	0.56	0.40	0.48
Problem 2	0.49	0.39	0.42	0.46	0.36	0.56	0.44	0.40	0.43

Measuring performance portability is thus not obvious - while the metric does distinguish between running the application on different problems, it does not consider compilers and parallel programming methods. In the simplest case, we can calculate the overall performance portability, by averaging across the three algorithms, and picking the best results for each hardware platform. Individual efficiencies and the metric is shown in Table II - Problem 2 has a lower value due to the worse performing Algorithm 3. It is clear that the OpenMP and the CUDA parallelisations are the best performing ones on CPUs and GPUs respectively - however, the implementation and the code for these two look very different. While KOKKOS does have a single source code, its performance is overall slightly lower - for Problem 1, only 6% (0.46), but for Problem 2, by 22% (0.36).

Productivity is the third cornerstone, and it is a key factor to how well a given codebase can adapt to new hardware architectures. From the perspective of performance portability, it is still possible to have a range of implementations for different target architectures - both in different parallel programming tool, and code specialisations using the same tool. However, such a codebase severly impacts productivity; e.g. making changes in an algorithm requires modifying all its different implementations. Excessively specialised code paths also impact the readability and understandability of the code. Thus, having a single codebase, with minimal clutter is desired. This of course presents a trade-off with performance portability.

TABLE III
DISTANCE MATRIX FOR DIFFERENT IMPLEMENTATIONS

	CUDA	ACC	OMP+off	KOK	SYCL	OMP
CUDA	0	1	1	1	1	1
OpenACC	1	0	0.06	1	0.46	0.06
OpenMP+off	1	0.06	0	1	0.45	0.04
KOKKOS	1	1	1	0	1	1
SYCL	1	0.46	0.45	1	0	0.46
OpenMP	1	0.06	0.04	1	0.46	0

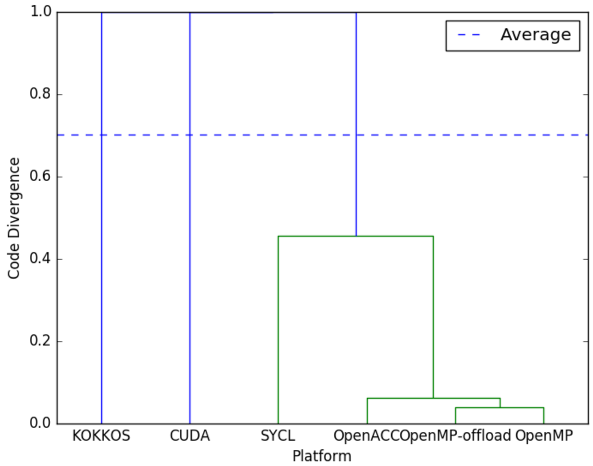

Fig. 4. Dendrogram of the code divergence between different implementations

TABLE IV
FRACTION OF PEAK PERFORMANCE, AND PERFORMANCE PORTABILITY METRIC OF INDIVIDUAL PROGRAMMING MODELS ON PROBLEM 1

	OpenMP	OpenACC	CUDA	SYCL	KOKKOS
ARM	0.52	X	X	X	0.55
Power9	0.54	X	X	X	0.51
KNL	0.36	X	X	X	0.35
Broadwell	0.51	X	X	0.45	0.46
Skylake	0.39	X	X	0.35	0.37
V100	0.66	0.67	0.71	0.68	0.62
P100	0.51	0.54	0.56	0.55	0.51
Portability	0.48	0.59	0.62	0.48	0.46

To have a grasp on productivity, we turn to the code divergence metric between different versions. The metric is commonly used to determine divergence in the same programming model (e.g. OpenMP), when targeting different architectures (e.g. different generations of GPUs). As in our implementation, we do no such specialisation (except for OpenMP and OpenMP offload), we instead study the divergence between different programming models.

Running the computational code (omitting the file read-in) through the Code Base Investigator, yields a distance matrix shown in Table III, and the dendrogram in Figure 4. Unsurprisingly, divergence is greatest with CUDA and KOKKOS, as both require a completely different source file - the former due to the outlined kernels, memory management, and kernel launches, and the latter due to the need for using the parentheses operator to access data. The second-largest divergence is with SYCL, which requires considerable setup code, but the body of the loops is the same as with OpenMP and OpenACC. Finally, OpenMP, OpenMP offload, and OpenACC are only different in the pragmas they use for data movement and the description of parallelism.

These results prompt the question of what performance and portability would be if we reduced code divergence. This is of course the main goal of portability libraries such as KOKKOS or RAJA. For our experiments, getting rid of the CUDA implementation would significantly reduce divergence - while it is the best performing version on GPUs, it only runs on GPUs. OpenMP and KOKKOS are the most obvious choices - while with OpenMP, its CPU and offload versions do not use the same pragmas, there is very little code divergence. If we were to rely solely on SYCL, that would remove divergence altogether, however currently the ARM, Power 9, and KNL platforms do not officially support it - something we expect to change in the future. Table IV (Problem 1) and Table V (Problem 2) display efficiencies and performance portability metrics for different programming models - omitting platforms that are not supported. Indeed, compared to the performance portability metric of 0.49/0.44, the OpenMP model sacrifices very little in performance (0.48/0.43), but supports all the platforms. Similarly, KOKKOS supports all platforms, at a slightly larger performance loss (0.46/0.36), particularly on Problem 2. SYCL has the same performance portability, though currently supports only a subset of platforms. While CUDA and OpenACC have better portability, they only support GPUs, which is not expected to change.

TABLE V
FRACTION OF PEAK PERFORMANCE, AND PERFORMANCE PORTABILITY METRIC OF INDIVIDUAL PROGRAMMING MODELS ON PROBLEM 2

	OpenMP	OpenACC	CUDA	SYCL	KOKKOS
ARM	0.49	X	X	X	0.45
Power9	0.39	X	X	X	0.42
KNL	0.42	X	X	X	0.22
Broadwell	0.46	X	X	0.39	0.40
Skylake	0.36	X	X	0.28	0.32
P100	0.56	0.50	0.56	0.56	0.47
V100	0.42	0.43	0.44	0.44	0.40
Portability	0.43	0.46	0.49	0.39	0.36

VI. CONCLUSIONS

We have carried out a thorough benchmarking of OpenMP, OpenMP offload, CUDA, OpenACC, SYCL, and KOKKOS parallelisation methods on a variety of hardware architectures and compilers, on three key multi-material kernels, and their variants. We demonstrated that the choice of algorithmic variant, compiler, runtime environment, and parallelisation approach is highly non-trivial, and may lead to significant differences in performance. In our study, while data structures are mostly incompatible with each other (linked lists vs. CSR), the fusion and fission algorithmic variants are not. Conversion between different multi-material data structures is usually prohibitively expensive, and compiling different algorithms with different compilers is also quite challenging - and sometimes not possible.

The availability of compilers is also an issue. Cray compilers are exclusively available on Cray machines, an IBM's XL compilers only on IBM systems. GCC is available on practically all platforms, and Intel compilers are also available in most clusters. There is a considerable push to integrate new parallelisations, and generally improve the performance of LLVM/Clang - it is already the compiler with the widest range of supported platforms - CUDA and OpenMP offload support are already officially supported, and Intel's SYCL implementation will eventually be pulled in. Its performance is already quite promising, and on par with others - except in the area of auto-vectorisation for CPUs, where Intel compilers are still the best by far.

OpenMP as a programming model has proved it can adapt to new hardware, though it, and its implementations, are always behind the cutting edge in terms of supporting new features. In our tests, its performance was very competitive to lower-level approaches, such as CUDA. The latest emerging technology, SYCL, is still in its early stages in terms of compiler and hardware support, but it has proven quite flexible, and has significant traction. The performance of current SYCL implementations is still lagging behind other more established programming models - and it struggles with the same problem as OpenCL did - portability does not mean performance portability.

Performance portability metrics were calculated, and are particularly interesting when contrasted with code divergence metrics - giving up a small amount of performance can significantly reduce divergence.

While maintainability and general readability can be significantly affected by a larger code divergence, another factor that has huge impact is the readability of the "science code" itself - which in our study was largely unchanged between parallelisations. Because of the different data structure variants and the algorithmic variants, as well as the complexity of handling mixed and pure cells slightly differently (not numerically, just in terms of accessing different arrays), this science code is also very hard to read and understand. This has prompted us to start developing a Domain Specific Library targeting multi-material data structures and algorithms [21], which abstracts the specific data structure away, significantly simplifying science code. We are currently working on generating code automatically that matches the hand-written implementations discussed in this paper.

Acknowledgment

We acknowledge help and development work by Gihan R. Mudalige, Attila Jász and Dániel Becker. István Reguly was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013). This work used the Isambard UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1)

References

[1] M. M. Waldrop, "The chips are down for moores law," *Nature News*, vol. 530, no. 7589, p. 144, 2016.

[2] "Vector neural network instructions enable int8 ai inference on intel architecture," Apr 2019. [Online]. Available: https://www.intel.ai/vnni-enables-inference/

[3] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018, pp. 522–531.

[4] C. J. Newburn, S. Dmitriev, R. Narayanaswamy, J. Wiegert, R. Murty, F. Chinchilla, R. Deodhar, and R. McGuire, "Offload compiler runtime for the intel® xeon phi coprocessor," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1213–1225.

[5] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating gpu threads for openmp 4.0 in llvm," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*. IEEE Press, 2014, pp. 12–21.

[6] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 859–870. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85

[7] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[8] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[9] R. Keryell, R. Reyes, and L. Howes, "Khronos sycl for opencl: A tutorial," in *Proceedings of the 3rd International Workshop on OpenCL*, ser. IWOCL '15. New York, NY, USA: ACM, 2015, pp. 24:1–24:1. [Online]. Available: http://doi.acm.org/10.1145/2791321.2791345

[10] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[11] R. D. Hornung and J. A. Keasler, "The raja portability layer: Overview and status."

[12] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, 2017.

[13] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov 2018, pp. 24–36.

[14] Intel, "Code base investigator." [Online]. Available: https://github.com/intel/code-base-investigator

[15] R. V. Garimella and R. W. Robey, "A comparative study of multi-material data structures for computational physics applications," Tech. Rep.

[16] S. Fogerty, M. Martineau, R. Garimella, and R. Robey, "A comparative study of multi-material data structures for computational physics applications," *Computers & Mathematics with Applications*, vol. 78, no. 2, pp. 565 – 581, 2019, proceedings of the Eight International Conference on Numerical Methods for Multi-Material Fluid Flows (MULTIMAT 2017). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0898122118303341

[17] I. Reguly, "Multi-material implementations." [Online]. Available: https://github.com/reguly/multimaterial

[18] A. Alpay, "hipsycl," Aug 2019. [Online]. Available: https://github.com/illuhad/hipSYCL

[19] C. Ltd., "Codeplay - computecpp," 2019. [Online]. Available: https://www.codeplay.com/products/computesuite/computecpp

[20] I. Z. Reguly, A.-K. Keita, and M. B. Giles, "Benchmarking the ibm power8 processor," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*. IBM Corp., 2015, pp. 61–69.

[21] D. Becker, I. Reguly, and G. Mudalige, "An abstraction for local computations on structured meshes and its extension to handling multiple materials," in *CNNA 2018; The 16th International Workshop on Cellular Nanoscale Networks and their Applications*. VDE, 2018, pp. 1–4.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: PERFORMANCE PORTABILITY OF MULTI-MATERIAL KERNELS

A. Abstract

This artifact comprises the source code, datasets, and build instructions on GitHub that can be used to reproduce our results presented in our P3HPC 2019 paper.

B. Description

1) Check-list (artifact meta information):

- **Algorithm: Multi-material kernels**
- **Program: C/C++/CUDA code**
- **Compilation: Intel 2018.2, Cray 8.7.9, GCC 8.2/7.3.1, LLVM/Clang 9.0**
- **Data set: either auto-generated, or available at https://github.com/lanl/MultiMatTest/blob/master/volfrac.dat.tgz**
- **Run-time environment: Operating systems described in Section IV or the paper**
- **Hardware: described in SectionIV of the paper**
- **Execution: using GNU make - `OMP_NUM_THREADS=XX CC=compiler make test_cpu` for OpenMP tests, `CC=compiler make test_acc` for OpenMP offload and OpenACC tests, `CC=compiler make test_sycl` for SYCL tests, and `CC=compiler make test_gpu` for CUDA tests.**
- **Output: Elapsed times and bandwidth for the algorithmic variants and inputs.**
- **Experiment workflow: clone sources from GitHub, select compiler (e.g. clang++), and run the make commands**
- **Experiment customization: Edit flags in Makefile**
- **Publicly available?: Yes**

2) How software can be obtained (if available): Available on GitHub: https://github.com/reguly/multimaterial

3) Hardware dependencies: Runs on any system with at least 10GB or RAM.

4) Software dependencies: Requires Linux, and at least one of the OpenMP-enabled compilers described in Section IV of the paper. KOKKOS tests require a KOKKOS installation.

5) Datasets: When the compiled binaries are executed with no, or 2 arguments, a "volfrac.dat" input file is expected, one can be downloaded from https://github.com/lanl/MultiMatTest/blob/master/volfrac.dat.tgz. When three extra arguments are provided, a random problem is generated with fractions for 2-material, 3-material, and 4-material cells respectively. For examples, see the Makefile.

C. Installation

No installation required.

D. Experiment workflow

After cloning the GitHub repository, use GNU make - `OMP_NUM_THREADS=XX CC=compiler make test_cpu` for OpenMP tests, `CC=compiler make test_acc` for OpenMP offload and OpenACC tests, `CC=compiler make test_sycl` for SYCL tests, and `CC=compiler make test_gpu` for CUDA tests. XX should be replaced with the number of cores to be used. Thread binding is done with numactl or taskset - please edit the $NUMA$ field in the Makefile accordingly. To set the compiler, use `CC=compiler`, which could be one of `g++, clang++, icpc, pgc++, xlc++, CC, syclcc-clang, compute++`.

E. Evaluation and expected result

The makefile procedure will run a set of tests with different algorithmic variants and the file input, as well as the random input. Performance is printed for the three kernels: timings as well as bandwidth. Bandwidth results are reported as a fraction of the peak in the paper - so the bandwidth output of the runs should be divided by the peak bandwidth of the given platform.

F. Experiment customization

Compiler flags can be customized in the Makefile.